Research - Kasteran* — Formal Verification of Compiler Correctness

Alpasan, Lois-Kleinner

2026

Abstract

Formal verification of compiler correctness aims to prove that a compiler's output code faithfully implements the semantics of its input program. This document surveys the landmark achievements in verified compilation—CompCert, CakeML, seL4, VeriFast, and Dafny—and examines the verification strategies employed: simulation relations, translation validation, and proof-carrying code. We discuss Kasteran*'s verification pipeline, which combines the CompCert-style verified backend for critical optimizations with translation validation for aggressive transforms and SMT-based checking for type-level properties.

Kasteran* — Formal Verification of Compiler Correctness

Academic Research Document © Lois-Kleinner & 0-1.gg 2026

Abstract

Formal verification of compiler correctness aims to prove that a compiler's output code faithfully implements the semantics of its input program. This document surveys the landmark achievements in verified compilation—CompCert, CakeML, seL4, VeriFast, and Dafny—and examines the verification strategies employed: simulation relations, translation validation, and proof-carrying code. We discuss Kasteran*'s verification pipeline, which combines the CompCert-style verified backend for critical optimizations with translation validation for aggressive transforms and SMT-based checking for type-level properties.

1. Introduction

Compiler bugs are particularly insidious: a single error in the compiler can introduce vulnerabilities, incorrect behavior, or performance regressions into every program compiled with it. The formal verification of compiler correctness—proving that the compiler's output preserves the source program's semantics—provides the strongest guarantee of compiler reliability. Kasteran* incorporates formal verification at multiple levels, from the type system (which verifies program properties statically) to the code generator (which is verified to produce correct machine code).

2. Historical Background

The dream of verified compilation dates to the 1960s. McCarthy and Painter's proof of a simple arithmetic expression compiler established the template: define the semantics of source and target

languages, then prove a simulation relation between them (McCarthy and Painter 1). John McCarthy's work on mathematical theory of computation laid the foundations for reasoning about program correctness (McCarthy 1).

The CompCert project, initiated by Xavier Leroy at INRIA, was the first to verify a realistic optimizing C compiler end-to-end (Leroy 1). CompCert is written in Coq and verified using Coq's proof assistant. The compiler comprises a front-end (parsing, type checking, translation to Cminor), a middle-end (optimization passes on Cminor and RTL), and a back-end (instruction selection, register allocation, code generation). Each pass is proved to preserve the semantics of the source program using simulation diagrams.

CakeML, developed at the University of Cambridge and Chalmers University, provides a verified ML implementation (Myreen et al. 1). Unlike CompCert, which is verified in Coq, CakeML is verified in HOL4. CakeML includes a bootstrapping proof: the compiler's correctness proof covers the entire compilation pipeline from CakeML source to machine code for ARM, x86-64, and RISC-V. The CakeML compiler is written in CakeML itself, and its correctness proof shows that the self-compiled compiler is correct.

The seL4 microkernel verification, while not a compiler, demonstrated the feasibility of verifying large, realistic system software (Klein et al. 1). seL4's verification in Isabelle/HOL proved functional correctness, security enforcement, and absence of common implementation errors (buffer overflows, null pointer dereferences, arithmetic errors). The seL4 verification established techniques—including formal specification, Hoare logic proofs, and C code verification using tools like AutoCorres—that are directly applicable to verification of compilers and runtime systems.

3. Technical Analysis

The central technique in verified compilation is the simulation proof. For each compilation pass from source language S to target language T, the verifier defines:

1. A state relation R between S-states and T-states.
2. A well-formedness condition W on S-states.
3. A proof that for any transition S $\rightarrow$ S (preserving W), there exists a transition T $\rightarrow^*$ T such that R(S , T) and R(S , T).

This "forward simulation" ensures that every behavior of the source program is preserved in the compiled program. For backward simulation (ensuring every behavior of the compiled program corresponds to some source behavior), a separate "backward simulation" proof is required.

CompCert uses a "lock-step" simulation for most passes: each source transition corresponds to exactly one target transition. For optimization passes that may skip or combine operations, the simulation is "star-step": a source transition may correspond to zero or more target transitions. The transitivity of simulation relations allows composing pass-level correctness proofs into an end-to-end correctness theorem.

Translation validation takes a different approach: rather than proving correctness once (a priori), the compiler generates a proof of correctness for each compilation instance (a posteriori). The validator checks the proof at compile time, rejecting incorrect transformations (Necula 1). This approach enables aggressive optimizations that would be difficult to verify a priori, at the cost of increased compilation time.

The Verified Software Toolchain (VST) applies separation logic to verify C programs against functional correctness specifications (Appel et al. 1). VST uses a Hoare logic embedded in Coq, with assertions expressed in separation logic extended with first-order logic and arithmetic. The verified C programs include data structures (red-black trees, hash tables), cryptographic algorithms (AES, SHA-256), and operating system components (page allocators, interrupt handlers).

4. Current State of the Art

The Verified Cminusminus project extends CompCert's verification to concurrent programs. The CompCertTSO model verifies compilation to the x86-TSO (Total Store Order) relaxed memory model, proving that the compiler does not introduce spurious data races or reorder memory operations unsoundly (Ševčík et al. 1).

The Vellvm project formalizes LLVM IR in Coq, providing a verified infrastructure for LLVM optimization passes (Zhao et al. 1). Vellvm's formal semantics cover the core LLVM IR, including SSA form, memory operations, and exception handling. The Alive project extends this direction with SMT-based verification of LLVM peephole optimizations (Lopes et al. 1).

VeriFast, developed at KU Leuven, is a program verifier for C and Java that uses symbolic execution and separation logic (Jacobs et al. 1). VeriFast is used in industrial contexts for verifying safety-critical software, demonstrating that formal verification tools can be practical for production code.

Dafny, developed by K. Rustan M. Leino at Microsoft Research, integrates verification into a programming language (Leino 1). Dafny programs include preconditions, postconditions, loop invariants, and termination metrics, which are verified by the integrated Boogie verifier and Z3 SMT solver. Dafny is used in production at Amazon Web Services for verifying critical AWS components.

5. Relevance to Kasteran*

Kasteran* adopts a pragmatic approach to formal verification. The compiler's type system serves as a lightweight verifier: many correctness properties—memory safety, data-race freedom, resource management—are enforced by the type system at compile time without requiring explicit verification annotations.

For properties beyond the type system's reach, Kasteran* provides verification annotations:

```
fn sort(arr: &mut [Int]) ensures sorted(arr) {
    // implementation
}
```

The compiler generates verification conditions from these annotations and attempts to discharge them using Z3. For annotations that cannot be automatically verified, the compiler produces a proof obligation that can be interactively discharged using an external proof assistant (Coq or Isabelle).

The compiler's optimization passes are partially verified. Passes proven correct by simulation (similar to CompCert) are used for the safe, in-order code path. Aggressive optimization passes (auto-vectorization, loop transformations) use translation validation: after applying the optimization, the compiler generates a verification condition and checks it with Z3. If the check fails, the compiler falls back to the unoptimized code.

The bootstrapping compiler—the version of Kasteran* written in Kasteran* itself—is verified using the CakeML approach. The verified Kasteran* spec in HOL4 defines the language's semantics, and the compiler's core passes are proved correct against this specification.

6. Future Directions

The grand challenge in verified compilation is "correctness with optimality." Current verified compilers guarantee that semantics are preserved but say nothing about code quality. Proving that generated code is optimal (e.g., minimal register pressure, optimal instruction selection) remains largely open. The Verified Optimization project at Carnegie Mellon aims to develop techniques for proving both correctness and optimality of compiler transformations.

Another frontier is the verification of just-in-time (JIT) compilers. JIT compilers generate code at runtime, making verification more challenging. The work on "verified JIT compilation" using Coq promises to extend formal verification to dynamic compilation scenarios (Myreen 1).

Works Cited

Appel, Andrew W., et al. "Verified Software Toolchain." *Proceedings of the 20th European Symposium on Programming*, 2011, pp. 1-17.

Jacobs, Bart, et al. "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java." *Proceedings of the 2011 NASA Formal Methods Symposium*, 2011, pp. 1-14.

Klein, Gerwin, et al. "seL4: Formal Verification of an OS Kernel." *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009, pp. 207-220.

Leino, K. Rustan M. "Dafny: An Automatic Program Verifier for Functional Correctness." *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, 2010, pp. 348-370.

Leroy, Xavier. "Formal Verification of a Realistic Compiler." *Communications of the ACM*, vol. 52, no. 7, 2009, pp. 107-115.

Lopes, Nuno P., et al. "Alive: A Domain-Specific Language for Automatic Verification of LLVM Optimizations." *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 1-12.

McCarthy, John. "A Basis for a Mathematical Theory of Computation." *Computer Programming and Formal Systems*, North-Holland, 1963, pp. 33-70.

McCarthy, John, and James Painter. "Correctness of a Compiler for Arithmetic Expressions." *Mathematical Aspects of Computer Science*, vol. 19, 1967, pp. 33-41.

Myreen, Magnus O. "Verified Just-in-Time Compiler Generation." *Journal of Automated Reasoning*, vol. 54, no. 3, 2015, pp. 215-239.

Myreen, Magnus O., et al. "A Verified ML Compiler and Its Machine-Code Semantics." *Journal of Automated Reasoning*, vol. 53, no. 3, 2014, pp. 213-247.

Necula, George C. "Translation Validation for an Optimizing Compiler." *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000, pp. 83-94.

Ševčík, Jaroslav, et al. "Relaxing Memory Models with Validated Compilation." *ACM Transactions on Programming Languages and Systems*, vol. 34, no. 4, 2013, pp. 1-42.

Zhao, Jianzhou, et al. "A Formal Semantics of LLVM IR in Coq." *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012, pp. 1-12.

Gordon, Michael J. C. "The Semantic Basis of Program Verification." *Logic of Programming and Calculi of Discrete Design*, 1987, pp. 1-10.

Dijkstra, Edsger W. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs." *Communications of the ACM*, vol. 18, no. 8, 1975, pp. 453-457.

Hoare, C. A. R. "An Axiomatic Basis for Computer Programming." *Communications of the ACM*, vol. 12, no. 10, 1969, pp. 576-580.

Floyd, Robert W. "Assigning Meanings to Programs." *Proceedings of Symposia in Applied Mathematics*, vol. 19, 1967, pp. 19-32.

Reynolds, John C. "Separation Logic: A Logic for Shared Mutable Data Structures." *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, 2002, pp. 55-74.

O'Hearn, Peter W. "Resources, Concurrency, and Local Reasoning." *Theoretical Computer Science*, vol. 375, no. 1-3, 2007, pp. 271-307.

Yang, Hongseok, and Peter W. O'Hearn. "A Semantic Basis for Local Reasoning." *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*, 2002, pp. 402-416.

Chlipala, Adam. "Formal Reasoning About Programs." *Foundations and Trends in Programming Languages*, vol. 9, no. 1, 2022, pp. 1-150.

Pierce, Benjamin C., et al. *Software Foundations*. Electronic textbook, 2023.

Krebbers, Robbert. "A Formal C Semantics in Coq." *Proceedings of the 2014 ACM SIGPLAN Conference on Certified Programs and Proofs*, 2014, pp. 1-12.